Accepted for publication in *IEEE Design and Test of Computers*.

SUAVE: Extending VHDL to Improve Support for Data Modeling*

Peter J. Ashenden
University of Adelaide

Philip A. Wilsey and Dale E. Martin
University of Cincinnati

Abstract

Designers are increasingly using VHDL for high-level modeling. However, their task is hindered by the lack of object-oriented and genericity features in the language. Experience in programming languages shows that these features significantly aid management of complexity in large designs and promote re-use of modules. SUAVE extends VHDL by adapting several object-oriented and genericity features from Ada-95. The extensions improve support for modeling in VHDL from system level down to gate level. This article describes the extensions and illustrates their use with examples.

Index Terms: VHDL, object-oriented, genericity, hardware modeling.

Introduction

VHDL [1, 6] is widely used by designers of digital systems for specification, simulation, and synthesis. Increasingly, designers are using VHDL at high levels of abstraction as part of the system-level design process. At this level of abstraction, designers describe the aggregate behavior of a system in a style similar to that of software. They model data in abstract form, rather than using any particular binary representation, and express functionality in terms of interacting processes that perform algorithms of varying complexity. They may defer determining which aspects of the modeled behavior are to be implemented as hardware subsystems, and which are to be implemented as software, until later in the design flow.

Designers using VHDL for high-level modeling can learn much from the experiences of the software engineering community. There, engineers have widely adopted object-oriented design and programming techniques for managing complexity through abstract data types (ADTs) and re-use. Programming languages include features for abstraction, encapsulation, inheritance, and genericity to support these techniques. Software engineers use the terms *object-based* to refer to a language that includes abstraction and encapsulation mechanisms, and *object-oriented* to refer to a language that additionally includes inheritance [10].

While VHDL is usable for modeling at the system level, it has some deficiencies that make the task more difficult than it need be. These difficulties center around language features (or lack of some features) for supporting complexity management. In particular, VHDL is currently somewhat less than object-based, as its encapsulation mechanism is weak; it is certainly not object-oriented, as it does not include any form of inheritance. While it does in-

* This work was partially supported by Wright Laboratory under USAF contract F33615-95-C-1638.

clude a mechanism for genericity, that mechanism is severely limited, allowing only parameterization of units by constant values. Thus, VHDL cannot readily be used to define secure, re-usable ADTs.

Our aim in the SUAVE (SAVANT and University of Adelaide VHDL Extensions) Project is to improve support for high-level modeling and re-use in VHDL. A number of previous proposals also address these goals (see sidebar). SUAVE extends the language with features for object-orientation and genericity, and generalizes some existing features. Extending VHDL in this way has the side-effect of improving its expressiveness at all levels of abstraction. In this article we describe our language extensions, argue for their need, and illustrate their use with examples. We describe the extensions in more detail in a separate report [3]. Most of the features that we have added to VHDL are adapted from Ada-95 [7] and are included largely for the same reasons that they are included in Ada-95 [4].

Design objectives

Our analysis of requirements for extending VHDL for system-level modeling [2] leads to the following design objectives:

1. to improve support for high-level behavioral modeling by improving encapsulation and information hiding capabilities and providing for hierarchies of abstraction,
2. to improve support for re-use and incremental development by allowing later bindings through type-genericity and dynamic polymorphism,
3. to preserve capabilities for synthesis and other forms of design analysis,
4. to support hardware/software co-design through improved integration with programming languages (e.g., Ada),
5. to support refinement of models through elaboration of components rather than through repartitioning, and
6. to preserve correctness of existing models within the extended language.

Since SUAVE is an extension of the existing VHDL language, it is important that the extensions integrate well with all aspects of the existing language. In order to achieve that aim, we also followed the design principals used during the restandardization of VHDL in 1993. They required preserving the core semantic concepts in the language: strong typing, scope of modeling, consistency, generality, intermixing of abstraction levels, and so on. Our goal was to preserve the "conceptual integrity" of the language. As Fred Brooks notes [5, page 49], "Conceptual integrity does require that a system reflect a single philosophy and that the specification as seen by the user flow from a few minds." We sought to design extensions were conceptually integrated with the existing language, rather than appearing as a disjoint set of features added *post hoc*.

Improved support for data modeling

A data type in VHDL is characterized by a set of values and a set of operations. The set of values is specified by a named type definition, for example, an array or record type definition.

The operations are specified as procedures and functions that take parameters of the named type. The problem with this simplified approach is that a user of the type can inadvertently (or deliberately) manipulate values of the named type directly, rather than by using the operations associated with the type. A better approach is to define an abstract data type (ADT). In this approach, the concrete details of the type definition are hidden from users of the ADT. Users may only use the ADT operations to manipulate values; they may not manipulate the values directly.

Encapsulation

VHDL currently includes the *package* feature, used to collect a set of declarations together. An ADT is declared in a high-level model by declaring the type and associated operations in a package. However, VHDL does not currently provide a mechanism to encapsulate the concrete type details and to hide them from ADT users. Thus, an ADT declared in this way is not secure.

SUAVE extends the type system and package feature of VHDL to improve facilities for defining ADTs by adopting features from Ada-95. A package may be divided into a *visible part*, containing the *private types* and the operations of the ADT, and a *private part*, containing the concrete details of the type. This meets the first of our design objectives: to improve encapsulation and information hiding. Packages may also be defined in any declarative region of a VHDL model, rather than just as a top-level library unit. This improves support for abstraction by allowing a designer to define an ADT locally in part of a model.

In adopting the Ada-95 features for private types into VHDL, we needed to make some minor changes. In particular, VHDL includes a restriction that signals may not include elements that are access values (pointers). This restriction prevents processes from communicating pointers to variables and thereby gaining uncontrolled shared access to the variables. Consequently, SUAVE requires that a private type whose implementation may contain an element of an access type be declared with the reserved words **access private**. This allows an analyzer to prevent such types from being used in signal declarations and preserves the ability to define ADTs whose concrete details remain hidden from the user.

The following package declaration illustrates the extensions in SUAVE. It defines an ADT for complex numbers.

```
package complex_numbers is

    type complex is private;

    constant i : complex;

    function "&" ( L, R : real ) return complex;
    function re ( C : complex ) return real;
    function im ( C : complex ) return real;
    function "abs" ( C : complex ) return real;
    function arg ( C : complex ) return real;

    function "+" ( L, R : complex ) return complex;
    function "–" ( L, R : complex ) return complex;
    function "*" ( L, R : complex ) return complex;
    function "/" ( L, R : complex ) return complex;
```

```
private

    type complex is
        record
            re, im : real;
        end record complex;

end package complex_numbers;
```

The part before the second keyword **private** is the visible part of the package, and the remainder is the private part. The type complex is declared as a private type in the visible part of the package. This means that a user of the package can declare objects of the type, but cannot operate on them directly. The user must manipulate them with the operations provided by the package. The fact that complex numbers are represented in Cartesian form is not visible to a user. Indeed, the representation could be changed without the user's source code being affected. Here is an example of declaration of complex number objects and use of the operations:

```
use complex_numbers.all;
signal a, b, sum : complex := 0.0 & 0.0;
signal enable : bit;
. . .
sum <= a + b after 10 ns when enable = '1' else
        0.0 & 0.0 after 10 ns;
```

Note that ADTs defined using private types in packages do not include automatic constructors or destructors. Instead, values of a private type are initialized using the default initial value for the type. Alternatively, ADT operations, such as the "&" operator shown in the above example, can be used to initialize objects when they are declared. An ADT can also define explicit destructor operations to be called by the user to reclaim dynamic storage. This model of initialization and finalization is in keeping with the existing storage management model of VHDL.

Type derivation and classes

Object-oriented languages support re-use and incremental development through the mechanism of inheritance. SUAVE adopts the inheritance mechanisms of Ada-95, based on the notion of type derivation. When a type is defined in a package, the operations defined in the package are called *primitive operations* of the type. A new type can be defined by deriving it from an existing type. The derived type inherits its set of values and primitive operations from the parent type. A primitive operation can be overridden by defining an alternative operation with the same name as an inherited operation, but with parameters of the derived type. Furthermore, the set of primitive operations can be augmented by defining new operations with parameters of the derived type. The details of SUAVE's inheritance mechanism are described below. To illustrate the inheritance features, we use them to define data types and operations representing a RISC instruction set. The hierarchy of instructions is illustrated in Figure 1.

As in Ada-95, a SUAVE record type can be declared *tagged*. This means that the representation of a value of the type includes a tag that identifies the specific type of the value. When a new type is derived from a tagged type, the new type can extend the set of inherited record

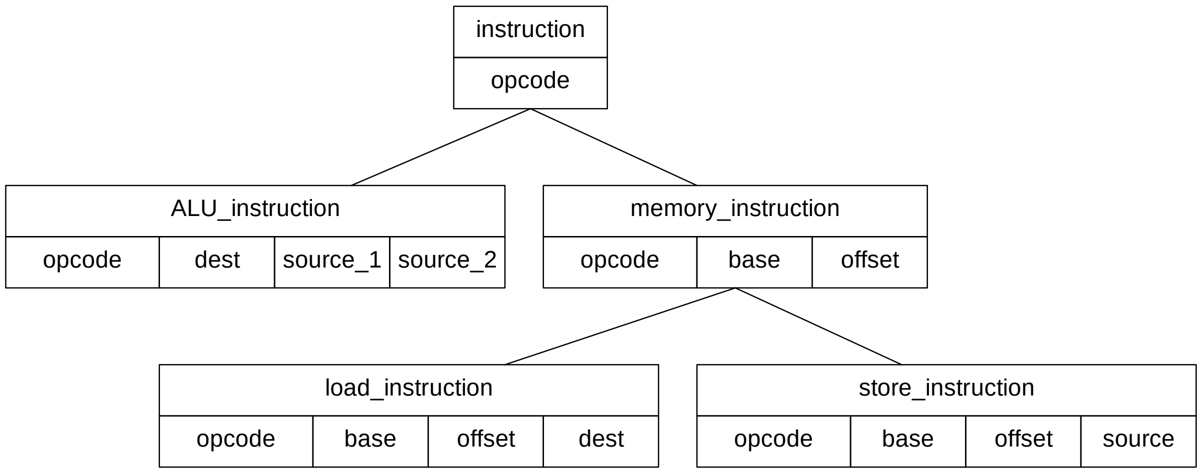

Figure 1. Inheritance hierarchy for a RISC computer instruction set.

elements with additional elements. This is the origin of the term “programming by extension.” Operations defined for the parent type are applicable to objects of the derived type, since the derived type objects have all of the record elements of the parent type. Overriding or new operations defined for the derived type are only applicable to objects of the derived type, since they may refer to elements in the extended type.

We start our illustration of the programming by extension approach by defining a root instruction type that includes just an opcode.

```
type instruction is
    tagged record
        opcode : opcode_type;
    end record instruction;
```

We also define three operations on this type: one to determine whether an instruction is privileged, one to disassemble an instruction into a textual representation, and one to perform the instruction.

```
function privileged ( instr : instruction;  mode : protection_mode ) return boolean;

procedure disassemble ( instr : instruction;  file output : text );

procedure perform ( instr : instruction );
```

The root type and operations can be used for instructions that are specified completely by the opcode. Next, we derive a new type to represent ALU instructions that have two source register numbers and a destination register number.

```
type ALU_instruction is new instruction with
    record
        dest, source_1, source_2 : register_number;
    end record ALU_instruction;
```

This type inherits the opcode element from the parent type and adds new elements for the register numbers. The type also inherits the operations. However, we override disassemble and perform by defining new versions for the derived type.

```
procedure disassemble ( instr : ALU_instruction;  file output : text );
procedure perform ( instr : ALU_instruction );
```

SUAVE also adopts the Ada-95 notion of *abstract types* and *abstract operations*. An abstract type cannot be used directly to define an object. Instead, a derived type must be used to instantiate an object whose interface follows that of the abstract type. Similarly, an abstract operation on a type is one that does not have an implementation, and so cannot be invoked. Instead, as new types are derived from abstract types, they also inherit the abstract operations for which implementations must be defined. The combination of these mechanisms allows definition of types that include common properties and operations, but which must be refined by derivation to types that represent concrete objects.

We illustrate use of abstract types and operations by defining types for load and store instructions using displacement addressing mode. We could derive the types from instruction, but that would involve duplicating those aspects dealing with the addressing mode. Instead, we define an abstract type for memory instructions, derived from instruction, that just includes the addressing mode aspects.

```
type memory_instruction is abstract new instruction with
    record
        base : register_number;
        offset : integer;
    end record memory_instruction;
```

We define an operation to calculate the effective address of a memory instruction and an abstract operation to perform a memory transfer. The effective_address_of operation is inherited for load and store instructions. It is not an abstract operation, since its implementation can be defined using the information in the memory_instruction type. The perform_memory_transfer operation, on the other hand, is abstract since its implementation varies between load and store instructions.

```
function effective_address_of ( instr : memory_instruction ) return natural;

procedure perform_memory_transfer ( instr : memory_instruction ) is abstract;
```

Next, we derive the specific types for load and store instructions and the overriding operations to perform the memory transfers. The load instruction type includes a destination register number, while the store instruction type includes a source register number.

```
type load_instruction is new memory_instruction with
    record
        dest : reg_number;
    end record load_instruction;

procedure perform_memory_transfer ( instr : load_instruction );

type store_instruction is new memory_instruction with
    record
        source : reg_number;
    end record store_instruction;

procedure perform_memory_transfer ( instr : store_instruction );
```

We can continue extending the type hierarchy in this manner to completely define the instruction set.

An important aspect of programming by extension is the use of *class-wide types*. Given a tagged type T, the class-wide type T'Class denotes T and all types derived directly or indirectly from it. SUAVE allows constants, signals, and dynamically allocated variables to be of class-wide types. Such objects are *polymorphic*, meaning that they can take on values of different specific types at different times. When an operation is applied to a polymorphic value, the tag of the value determines which particular version of the operation to invoke. This is called *run-time binding*, or *dynamic dispatching*. An operation that has a parameter of a class-wide type is called a *class-wide* operation. The parameter is polymorphic, so dynamic dispatching may be required when the operation calls further operations on the parameter.

We illustrate use of class-wide types and operations by defining an operation to execute any type of instruction in the instruction set.

```
procedure execute ( instr : instruction'Class );
```

The parameter instr is of a class-wide type, so on different calls, it may be of any of the specific types derived from instruction. The body of the operation can only assume that the parameter has those elements and operations applicable to the type instruction. Hence, it may only refer to the opcode element and the privileged, disassemble and perform operations, for example:

```
procedure execute ( instr : instruction'Class ) is
begin
    disassemble ( instr, trace_file );
    if privileged(instr) and execution_mode = user then
        handle_privilege_violation;
    else
        perform(instr);
    end if;
end procedure execute;
```

While there are versions of disassemble and perform for the instruction type, types derived from instruction override the operations. The tag of the value passed as instr is used to identify the type of that value, and thus which versions of disassemble and perform to invoke.

We can use the class-wide type instruction'class to define other kinds of objects, apart from parameters of operations. For example, the following entity declaration describes the interface of an instruction register that stores an instruction and that can jam a NOP instruction in place of the stored instruction. The last two register ports are polymorphic signal objects.

```
entity instruction_reg is
    port ( load_enable : in bit;  jam_nop : in bit;
            instr_in : in instruction'class;
            instr_out : out instruction'class );
end entity instruction_reg;
```

The architecture body for the instruction register, shown below, contains a process that represents the stored value using a local variable. Since only a dynamically allocated variable can be polymorphic in SUAVE, we use an access type for the variable.

```
architecture behavioral of instruction_reg is
begin

    store : process ( load_enable, jam_nop, instr_in ) is
        constant nop_instruction : instruction := instruction'(opcode => op_nop);
        constant undef_instruction : instruction := instruction'(opcode => op_undef);
        type instruction_ptr is access instruction'class;
        variable stored_instruction : instruction_ptr := new undef_instruction;
    begin
        if jam_nop = '1' then
            stored_instruction := new nop_instruction;
        elsif load_enable = '1' then
            stored_instruction := new instr_in;
        end if;
        instr_out <= stored_instr.all;
    end process store;

end architecture behavioral;
```

Interaction between encapsulation and inheritance

The extensions in SUAVE include features for combining encapsulation and inheritance. This allows definition of ADTs that can be extended without revealing their concrete type details to ADT users. As in Ada-95, a private type can be declared tagged so that it can be extended, and the extension can be declared private. We illustrate these mechanisms using an ADT representing tokens in an uninterpreted queuing model. We define a basic token type in a package as follows.

```
package tokens is

    type token is tagged private;

    function new_token return token;
    impure function token_age ( T : token ) return delay_length;
    . . .
private

    type token is
        tagged record
            id : natural;
            creation_time : time;
        end record token;

end package tokens;
```

This allows us to derive a type without knowing the concrete details of the parent, for example:

```
type colored_token is new token with
    record
        color : color_type;
    end record;

function new_token ( new_color : color_type := default_color ) return colored_token is
begin
    return ( token'(new_token) with color => new_color );
end function new_token;

variable next_token : colored_token := new_token;
variable subsequent_token : colored_token := new_token(red);
```

The expression in the return statement is an *extension aggregate* that extends a value of the parent type token with values required for the extension elements in the type colored_token.

As an illustration of a private extension, we define a new ADT for tokens that trace their history of flow in a queuing network. The new ADT is derived from the token ADT.

```
package traceable_tokens is
    type traceable_token is new token with private;
    function new_token return traceable_token;
    . . .
private
    type history_queue_type is ...
    type traceable_token is new token with
        record
            history_queue : history_queue_type;
        end record traceable_token;
end package traceable_tokens;
```

The only details visible to a user are that the traceable_token type is derived from the token type, and the operations available on the types.

Genericity

Re-use of a module can be improved by making it applicable in a wider set of contexts, for example, by making it more generic. VHDL currently includes the mechanism of generic constants that allows the designer to parameterize components and entities with formal constants. The designer specifies actual constants when instantiating components and when binding entities to component instances. The generic constant mechanism is widely used to specify timing parameters and array port sizes, among other things.

SUAVE extends the generic mechanism of VHDL to improve support for re-use. There are two main aspects to the extension, both adopted from Ada-95. First, in addition to components and entities, subprograms and packages are allowed to have generic interface clauses. Second, generic interface clauses can include formal types, making generic units reusable for a variety of different types. Generic interface clauses can also include formal subprograms and formal packages. We illustrate use of the genericity features in SUAVE with a number of examples.

A generic multiplexer

The first example, a generic multiplexer, shows how the simple extension of generic types significantly enhances re-use of design entities. The multiplexing operation simply involves choosing between two inputs, and does not depend on the type of the inputs. Hence, we can describe a generic multiplexer as follows:

```
entity generic_multiplexer is
    generic ( type data_type is private );
    port ( control : in bit;  in0, in1 : in data_type;  data_out : out data_type );
end entity generic_multiplexer;
```

The type of the data is denoted by the formal type data_type. The specification **is private** means that the actual type can be any type that allows assignment. Thus the multiplexer can be implemented with the following architecture body.

```
architecture data_flow of generic_multiplexer is
begin

    with control select
        data_out <= in0 when '0',
                    in1 when '1';

end architecture data_flow;
```

We can instantiate the generic multiplexer to select between integer inputs, for example, by associating the type integer with the formal type:

```
int_mux : entity work.generic_multiplexer(data_flow)
            generic map ( data_type => integer );
            port map ( . . . );
```

A generic sets package

The second example is a package that defines an ADT for sets of homogeneous elements.

```
package sets is
    generic ( type element_type is private );

    type set is access private;

    constant empty_set;

    procedure copy ( from : in set;  to : out set );

    function "+" ( R : element_type ) return set;                    -- singleton set
    impure function "+" ( L : set;  R : element_type ) return set;   -- add to set
    impure function "+" ( L : element_type;  R : set ) return set;   -- add to set
    . . .
private

    type element_node;
    type element_ptr is access element_node;
    type element_node is
        record
            next_element : element_ptr;
            value : element_type;
        end record element_node;
    type set is new element_ptr;

end package sets;
```

The formal type element_type denotes the type of element to be included in a set. The concrete type definition for a set and the operations on sets make use of the formal type name. A generic package such as this is a template for a package. We must instantiate it, associating an actual type with the formal type, to create a package whose declarations we can use, for example:

```
type test_vector is . . .
package test_sets is new sets generic map ( element_type => test_vector );
use test_sets.all;

variable tests_to_perform : test_sets.set := empty_set;
. . .

test_to_perform := test_to_perform + new_test;
```

Generic mix-in inheritance

The third example is a variant of the memory addressing instruction type shown above, and illustrates the interaction between genericity and inheritance features. There may be several addressing modes in the instruction set, each of which can be used in different kinds of instructions. It would be convenient to specify aspects of each addressing mode as a class that can be inherited using *mix-in* inheritance [9]. In languages such as C++, multiple inheritance can be used for this purpose. In Ada-95 and SUAVE, multiple inheritance is not needed. Instead, we use formal derived types to achieve the same effect. For example, we can define a mix-in package for an indexed addressing mode as follows:

```
package indexed_addressing_mixin is
    generic ( type parent_instruction is abstract new instruction with private );
    type indexed_instruction is new parent_instruction with
        record
            index_base, index_offset : register_number;
        end record indexed_instruction;
    function effective_address ( instr : indexed_instruction ) return address;
end package indexed_addressing_mixin;
```

The reserved words **with private** in the formal type specify that the type instruction is a specific tagged type and that parent_instruction is derived from it. Now suppose we derive a type load_instruction from type instruction as follows:

```
type load_instruction is abstract new instruction with
    record
        dest : register_number;
    end record load_instruction;
```

We can derive an indexed version of this instruction through instantiations of the indexed_addressing_mixin package, as follows:

```
package indexed_loads is
    new indexed_addressing_mixin generic map ( parent_instruction => load_instruction );
alias indexed_load_instruction is indexed_loads.indexed_instruction;
```

A generic shift register

The fourth example, a generic shift register, illustrates the use of different kinds of formal types in a generic interface. A shift register stores an array of elements, and on each clock, shifts elements along one position to include a new element. The entity interface includes formal types to define the array structure, as follows:

```
entity shift_register is
    generic ( type index_type is (<>);
              type element_type is private;
              type vector is array ( index_type range <> ) of element_type );
    port ( clk : in bit;
           data_in : element_type;
           data_out : vector );
end entity shift_register
```

The notation “(<>)” in the index type indicates that the actual index type must be a discrete type. The element type may be any type that allows assignment, and the vector type must be an unconstrained array. The architecture body is:

```
architecture behavioral of shift_register is
begin

    shift_behavior : process is
        constant data_low : index_type := data_out'low;
        constant data_high : index_type := data_out'high;
        type ascending_vector is array ( data_low to data_high ) of element_type;
        variable stored_data : ascending_vector;
    begin
        data_out <= stored_data;
        wait until clk = '1';
        stored_data(data_low to index_type'pred(data_high))
            := stored_data(index_type'succ(data_low) to data_high);
        stored_data(data_high) := data_in;
    end process shift_behavior;

end architecture behavioral;
```

The process makes use of the fact that the index type is a discrete type, by using values of the type in array bounds and by referring to the 'pred and 'succ attributes. A model might instantiate the design entity as follows:

```
signal master_clk, carry_in : bit;
signal result : bit_vector(15 downto 8);

bit_vector_shifter : entity work.shift_register(behavioral)
    generic map ( index_type => natural, element_type => bit, vector => bit_vector )
    port map ( clk => master_clk, data_in => carry_in,  data_out => result );
```

A generic lookup table package

The final example of the use of extended generics illustrates the use of formal subprograms in a generic interface. The following package defines an ADT for lookup tables of elements. Each element is identified by a key value. The element and key types are formal types of the package. In order to implement table operations, the package needs a function to determine the key of a stored element, and a function to compare keys. These two functions are included in the generic interface as formal subprograms. The notation "<>" associated with the comparison function means that the default function to use when the package is instantiated is the "<" function that is visible at the point of instantiation.

```
package lookup_tables is
    generic ( type element_type is private;
              type key_type is private;
              function key_of ( E : element_type ) return key_type;
              function "<" ( L, R : key_type ) return boolean is <> );

    type lookup_table is limited access private;

    procedure lookup ( table : in lookup_table;  lookup_key : in key_type;
                       element : out element_type;  found : out boolean );

    procedure search_and_insert ( table : in lookup_table;  element : in element_type;
                                  already_present : out boolean );

    procedure traverse
        generic ( procedure action ( element : in element_type ) )
        ( table : in lookup_table );

private
```

```
        type tree_record;
        type tree_ptr is access tree_record;
        type tree_record is
            record
                left_subtree, right_subtree : tree_ptr;
                element : element_type;
            end record tree_record;

        type lookup_table is new tree_ptr;

    end package lookup_tables;
```

The reserved word **limited** in the private type declaration for lookup_table specifies that the user may not perform assignment on lookup table values. We restrict lookup tables in this way, since they are implemented as pointers to binary search trees. Predefined assignment would simply copy the pointer, not the whole lookup table.

The procedure traverse illustrates the use of formal subprograms. This procedure performs an in-order traversal of a lookup table, applying the action procedure to each element. Once the user instantiates the package, they may also instantiate the traversal procedure with different actual action procedures to perform different actions on tables.

An outline of the package body is as follows:

```
    package body lookup_tables is

        procedure lookup ( table : in lookup_table;  lookup_key : in key_type;
                                element : out element_type;  found : out boolean ) is
            variable current_subtree : tree_ptr := tree_ptr(table);
        begin
            found := false;
            while current_subtree /= null loop
                if lookup_key < key_of( current_subtree.element ) then
                    current_subtree := current_subtree.left_subtree;
                elsif key_of( current_subtree.element ) < lookup_key then
                    current_subtree := current_subtree.right_subtree;
                else
                    found := true;
                    element := current_subtree.element;
                    return;
                end if;
            end loop;
        end procedure lookup;

        procedure search_and_insert ( table : in lookup_table;  element : in element_type;
                                              already_present : out boolean ) is ...

        procedure traverse
            generic ( procedure action ( element : in element_type ) )
            ( table : in lookup_table ) is
            alias tree is tree_ptr(table);
        begin
            if tree = null then
                return;
            end if;
            traverse ( lookup_table(tree.left_subtree) );
            action ( tree.element );
            traverse ( lookup_table(tree.right_subtree) );
        end procedure traverse;
```

```
end package body lookup_tables;
```

The bodies of the operations simply call the formal subprograms. The effect is to invoke the actual subprogram associated with the formal in the generic map of the package instance.

To illustrate use of the lookup table package, suppose we require a lookup table of test patterns that use character strings as keys. We instantiate the package as shown below. Since the predefined function "<" operating on strings is visible at the point of instantiation, it is used as the actual function for the formal function "<".

```
type test_pattern_type is . . .
function test_id_of ( test_pattern : in test_pattern_type ) return string;

package test_pattern_tables is
    new lookup_tables generic map ( element_type => test_pattern_type,
                                    key_type => string,
                                    key_of => test_id_of );
```

Note that, had the user overloaded the "<" operator on strings at the point of instantiation of the package, the overloaded version would be used. The user might overload the operator to use a different ordering for keys.

We can use the traversal procedure to count the number of elements in a table by instantiating it as follows:

```
variable count : natural := 0;

procedure count_a_test_pattern ( test_pattern : in test_pattern_type ) is
begin
    count := count + 1;
end procedure count_a_test_pattern;

procedure count_test_patterns is
    new test_pattern_tables.traverse generic map ( action => count_a_test_pattern );
```

We call the instantiated traversal function with a test pattern lookup table as a parameter, as follows:

```
variable patterns_to_apply : test_pattern_tables.lookup_table;
. . .
count_test_patterns ( patterns_to_apply );
```

Conclusions

In this article we have described our extensions to VHDL to improve its support for data modeling across the spectrum, from high-level behavioral modeling to gate-level modeling. We have drawn heavily on the approach used to extend Ada. This is appropriate, since the original definition of VHDL drew heavily on Ada. Thus, our extensions integrate well with the existing VHDL language, preserving its conceptual integrity.

Our extensions achieve the design objectives described earlier in this article. The first two objectives are directly achieved by the additional language features we have adopted from Ada-95. Considering the third objective, we note that synthesis is performed on an elaborated design. At this point in design processing, generic units have been instantiated with actual types substituted for formal types. Hence, analysis for synthesis is not affected by the SUAVE extensions for genericity. Similarly, the extensions for encapsulation, information hiding and type hierarchies relate to static semantic analysis, which occurs before analysis for synthesis. Hence these extensions do not impinge upon synthesis. The only exceptions are polymorphic data and dynamic dispatching. While it is, in principle, possible to synthesize hardware for these language features, it is likely that they would be excluded from synthesis subsets in the near future. This situation is analogous to the exclusion of features such as access types and dynamic allocation of objects from synthesis subsets. Other forms of analysis that are performed on models include formal verification of functionality and property checking. The SAUVE extensions require the formal bases for these types of analysis to be extended to deal with the semantics of the added language features.

The SUAVE extensions achieve the fourth design objective by bringing the type domain of VHDL closer to those of programming languages such as Ada-95, C++ and Java. Hence functionality that may ultimately be implemented in software is more readily expressed using the SUAVE extensions. This makes the language more suitable for use in a co-design environment.

We achieve the fifth design objective by retaining the concept of partitioning a design using process statements, entities and component instantiation. The high-level behavior of a module is described using process statements within an architecture of an entity. An instance of the entity forms an architectural partition (*c.f.* a block in an architectural block diagram). The partition may be refined by replacing the behavioral architecture with a structural architecture containing instances of more primitive entities. The SUAVE extensions described in this article do not modify this approach to refinement. Compare this with language extensions that adopt a monitor-style approach to expressing behavior (the third area discussed in the sidebar). Refining the behavior of a system expressed in that approach involves discarding the data-centric form and repartitioning into a process-centric form.

Lastly, we achieve the sixth design objective by ensuring that our extensions form a superset of existing language features. The only caveat is that we have introduced four new reserved words (**abstract**, **limited**, **private** and **tagged**). It is unlikely that an existing model would use any of these as a declared identifier, however, if it does, the model would not be correct in the extended langauge.

Space has prevented us from describing the full details of our extensions here, however, we hope that the reader has gained an appreciation of the style and capabilities of the extensions. We have specified the extensions in full elsewhere [3].

While we have addressed extensions to improve data modeling capabilities in VHDL, we have not in this article addressed the issues of concurrency and communication. One of the difficulties that has been identified with using VHDL for design at the system level is the concrete nature of communication between subsystems. Currently, each subsystem comprises one or more processes that communicate using signals. Communication events are scheduled at specific times, and a receiver must respond to an event when it occurs, or else miss the event. Hence, there is no notion of synchronization as is seen in conventional concurrent programming or system description languages. If more specific synchronization is required, it can be achieved either by implementing a two-way communication protocol over signals, or by instantiating communication intermediaries, such as explicit message queues. Either approach detracts from the abstraction of communication required at the system level of description.

In ongoing work, we are investigating improved support for system level modeling by providing a more abstract mechanism for communication and synchronization between processes and by generalizing the process model to allow dynamic instantiation and termination of processes. Such extensions to VHDL make it more suitable for describing systems before partitioning into hardware and software implementations, and make it more feasible to use VHDL uniformly throughout the design flow.

Additional work is in progress to implement the SUAVE extensions within the framework of the SAVANT project [8]. This work will result in compiler and simulation tools that will be used to develop system-level models for real, large designs. This work will provide valuable feedback on the use of language in real design flows, and will allow quantification of the implementation costs of the extensions.

References

[1] P. J. Ashenden, *The Designer's Guide to VHDL*. San Francisco, CA: Morgan Kaufmann, 1996.

[2] P. J. Ashenden and P. A. Wilsey, *Principles for Language Extension to VHDL to Support High-Level Modeling*, Dept. Computer Science, University of Adelaide, Technical Report TR-03/97, ftp://ftp.cs.adelaide.edu.au/pub/VHDL/TR-principles.ps, 1997.

[3] P. J. Ashenden, P. A. Wilsey, and D. E. Martin, *SUAVE: A Proposal for Extensions to VHDL for High-Level Modeling*, Dept. Computer Science, University of Adelaide, Technical Report TR-97-07, ftp://ftp.cs.adelaide.edu.au/pub/VHDL/TR-extensions.pdf, 1997.

[4] J. Barnes, Ed. *Ada 95 Rationale*, vol. 1247. Berlin, Germany: Springer-Verlag, 1997.

[5] F. P. Brooks, Jr., *The Mythical Man-Month*, Anniversary ed. Reading, MA: Addison-Wesley, 1995.

[6] IEEE, *Standard VHDL Language Reference Manual*. Standard 1076-1993, New York, NY: IEEE, 1993.

[7] ISO/IEC, *Ada 95 Reference Manual*. International Standard ISO/IEC 8652:1995 (E), Berlin, Germany: Springer-Verlag, 1995.

[8] MTL Systems Inc., *Standard Analyzer of VHDL Applications for Next-generation Technology (SAVANT)*. MTL Systems, Inc, http://www.mtl.com/projects/savant/, 1996.

[9] A. Taivalsaari, "On the Notion of Inheritance," *ACM Computing Surveys*, vol. 28, no. 3, pp. 438–479, 1996.

[10] P. Wegner, "Dimensions of Object-Based Language Design," *ACM SIGPLAN Notices*, vol. 22, no. 12, *Proceedings of OOPSLA '87*, pp. 168–182, 1987.

Sidebar

Related proposals for extending VHDL

A number of previous projects have devised object-oriented extensions to VHDL. We can classify them according to the issues they address. Some proposals simply provide object-oriented features for data modeling, similar to the features included in programming languages. Of these proposals, a number [4, 5, 12, 15, 16] extend VHDL by adding *classes*, as in C++ and Java. Others [11, 14] adopt the *programming by extension* approach of Oberon-2 and Ada-95. We have pursued the latter approach in SUAVE. Were we to include a class construct, we would replicate many of the encapsulation features already provided by the package construct of VHDL. Furthermore, dealing with the different kinds of storage objects in VHDL (variables and signals) is more complex when classes are used. Thus, the programming by extension approach yields a simpler, more consistent extension to the language [2].

A second area that some proposals address is re-use of design entities. In VHDL, an entity declaration describes a system's interface in terms of its connection ports, and an architecture body describes an implementation in terms of signals and concurrent statements. Proposals addressing entity re-use view an entity as a class, and allow a new entity to be derived from a parent entity [6, 11-13]. The derived entity inherits the ports of the parent, and can add further ports. Furthermore, an architecture can be derived from a parent architecture, inheriting the parents statements, and augmenting them to define additional functionality. SUAVE does not include this form of inheritance because, in practice, the amount of re-use that it affords is very limited, and the added complexity in the language is not warranted.. Behavioral models typically express functionality in one or a small number of process statements. Refining the functionality using the proposed approaches involves overriding the process statements rather than re-using them. More significant re-use comes from including instances of design entities in new designs, thus forming compositional hierarchies rather than classification hierarchies. VHDL already supports this form of re-use well.

A third area addressed by a number of previous proposals is abstract system-level modeling. These proposals [4, 5, 12, 15, 16] appeal to the object-oriented paradigm by adding procedural operations to the interface of a design entity. An architecture body implementing the entity includes data items (variables or signals) that are accessible to the procedural operations. Thus, these proposals view a system as a collection of objects interacting through monitor calls, with some form of concurrency control managing access to data encapsulated within architecture bodies. Unfortunately, there are some serious deficiencies in these proposals [1]. Furthermore, their adoption of the monitor paradigm is not supported by an analysis of requirements, but occurs as an extension of a class-based solution to data modeling. We argue that a monitor-based approach is generally inappropriate for system-level modeling [3]. It appears that many other system-level language designers agree—Statecharts [7], Estelle [9], SDL [10] and CSP [8] all use various forms of message passing.

References

[1] P. J. Ashenden and P. A. Wilsey, “Considerations on Object-Oriented Extensions to VHDL,” *Proceedings of VHDL International Users Forum Spring 1997 Conference*, Santa Clara, CA, pp. 109–118, 1997.

[2] P. J. Ashenden and P. A. Wilsey, “A Comparison of Alternative Extensions for Data Modeling in VHDL,” *Proceedings of Hawai'i International Conference on System Sciences*, Kona, Hawai'i, 1998.

[3] P. J. Ashenden and P. A. Wilsey, “Considerations on System-Level Behavioural and Structural Modeling Extensions to VHDL,” *Proceedings of VHDL International Users Forum Spring 1998 Conference*, Santa Clara, CA, 1998.

[4] J. Benzakki and B. Djaffri, “Object Oriented Extensions to VHDL: the LaMI Proposal,” *Proceedings of Conference on Hardware Description Languages '97*, Toledo, Spain, pp. 334–347, 1997.

[5] D. Cabanis and S. Medhat, “Classification-Orientation for VHDL: A Specification,” *Proceedings of VHDL International Users Forum Spring '96 Conference*, Santa Clara, CA, pp. 265–274, 1996.

[6] W. Ecker, “An Object-Oriented View of Structural VHDL Description,” *Proceedings of VHDL International Users Forum Spring '96 Conference*, Santa Clara, CA, pp. 255–264, 1996.

[7] D. Harel, “Statecharts: A Visual Formalism for Computer Systems,” *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, 1987.

[8] C. A. R. Hoare, “Communicating Sequential Processes,” *Communications of the ACM*, vol. 21, no. 11, pp. 934–941, 1978.

[9] ISO, *Estelle: A Formal Description Technique Based on an Extended State Transition Model*. Draft International Standard 9074, 1987.

[10] ITU, *Specification and Description Language (SDL)*. Revised Recommendation Z.100, 1992.

[11] M. T. Mills, *Proposed Object Oriented Programming (OOP) Enhancements to the Very High Speed Integrated Circuits (VHSIC) Hardware Description Language (VHDL)*, Wright Laboratory, Dayton, OH, Tech. Report WL-TR-5025, 1993.

[12] M. Radetzki, W. Putzke, W. Nebel, S. Maginot, J.-M. Bergé, and A.-M. Tagant, “VHDL Language Extensions to Support Abstraction and Re-Use,” *Proceedings of Workshop on Libraries, Component Modeling, and Quality Assurance*, Toledo, Spain, 1997.

[13] C. R. Ramesh, “Object Orienting VHDL for Component Modeling,” *Proceedings of VHDL International Users Forum Fall '94 Conference*, McLean, VA, pp. 5.17–5.28, 1994.

[14] G. Schumacher and W. Nebel, “Inheritance Concept for Signals in Object-Oriented Extensions to VHDL,” *Proceedings of Euro-DAC '95 with Euro-VHDL '95*, Brighton, UK, pp. 428–435, 1995.

[15] S. Swamy, A. Molin, and B. Covnot, “OO-VHDL: Object-Oriented Extensions to VHDL,” *IEEE Computer*, vol. 28, no. 10, pp. 18–26, 1995.

[16] J. C. Willis, S. A. Bailey, and R. Newschutz, “A Proposal for Minimally Extending VHDL to Achieve Data Encapsulation Late Binding and Multiple Inheritance,” *Proceedings of VHDL International Users Forum Fall ’94 Conference*, McLean, VA, pp. 5.31–5.38, 1994.